title: Sending Bundles

Getting our trade ready

After we find a transaction that touches the trading pair we're targeting, we need to calculate how many tokens we should expect to receive. In this project, we are specifically *buying* DAI with ETH (technically WETH, because Uniswap only trades ERC-20 tokens), so we want the price of DAI/WETH to be as low as possible. In code, we define our price requirements in terms of the *amount of tokens we receive* from the trade — so we want the *highest* amount of tokens possible, but we have a definite minimum (1800 DAI).

In terms of a limit order, we're saying that we want our order to be filled when the price is *at most* 1800 DAI, but if the price is lower, then we want to fill at the lower price, which yields more DAI per WETH.

To find our token's market price, we simulate our trade by calling the `swapExactTokensForTokens` function with a static call. A static call simply simulates the transaction, so we can see what it would do if we were to actually send it. We set our buy/sell amounts that we defined earlier to see how much we'd get from the swap. Add this function to your code — we'll add it to our main function later.

src/index.ts

tsx async function getBuyTokenAmountWithExtra() { const resultCallResult = await uniswapRouterContract .swapExactTokensForTokens .staticCallResult(SELL_TOKEN_AMOUNT, 1n, [SELL_TOKEN_ADDRESS, BUY_TOKEN_ADDRESS], executorWallet.address, 9999999999n) const normalOutputAmount = resultCallResult[0][1] const extraOutputAmount = normalOutputAmount * (10000n + DISCOUNT_IN_BPS) / 10000n return extraOutputAmount }

The minimum amount we can expect to receive from a swap is defined by `normalOutputAmount`. Then, we calculate how much we'd get if with a 40 basis-points discount, which we should expect if we successfully backrun a transaction that shifts the price in our favor, and assign this value to `extraOutputAmount`.

When we detect a new transaction, we'll need to check the going price and set up our trade accordingly. If the price is lower than our target, and because we're trying to *buy* tokens, we want to make sure our trade expects more tokens out; as many as we can get at the lower price with our fixed sell amount (the ETH we'll spend to buy the tokens). If the price is higher than our target, then we'll just set the expected output to the minimum amount we'd expect to if the price were at our target, in hopes that the transaction we backrun will move the price enough for us to make a trade.

Let's add a couple more functions to implement this logic:

src/index.ts

```tsx async function getSignedBackrunTx( outputAmount: bigint, nonce: number ) { const backrunTx = await uniswapRouterContract.swapExactTokensForTokens.populateTransaction(SELL_TOKEN_AMOUNT, outputAmount, [SELL_TOKEN_ADDRESS, BUY_TOKEN_ADDRESS], executorWallet.address, 9999999999n) const backrunTxFull = { ...backrunTx, chainId: 1, maxFeePerGas: MAX_GAS_PRICE * GWEI, maxPriorityFeePerGas: MAX_PRIORITY_FEE * GWEI, gasLimit: TX_GAS_LIMIT, nonce: nonce } return executorWallet.signTransaction(backrunTxFull) }

async function backrunAttempt( currentBlockNumber: number, nonce: number, pendingTxHash: string ) { let outputAmount = await getBuyTokenAmountWithExtra() if (outputAmount < BUY_TOKEN_AMOUNT_CUTOFF) { console.log(Even with extra amount, not enough BUY token: ${ outputAmount.toString() }. Setting to amount cut-off) outputAmount = BUY_TOKEN_AMOUNT_CUTOFF } const backrunSignedTx = await getSignedBackrunTx(outputAmount, nonce) try { const sendBundleResult = await mevshare.sendBundle({ inclusion: { block: currentBlockNumber + 1 }, body: [ { hash: pendingTxHash }, { tx: backrunSignedTx, canRevert: false } ] },) console.log('Bundle Hash: ' + sendBundleResult.bundleHash) } catch (e) { console.log('err', e) } } ```

The `getSignedBackrunTx` function creates the transaction we'll send to execute our trade on Uniswap. We set a fixed gas price here for simplicity. If you prefer, you could replace the with dynamic fees that track the base fee of the chain. But constant gas prices may work better if you don't want to spend a lot on gas, and don't mind having to wait if the network's base fee exceeds your settings.

The `backrunAttempt` function defines our price requirement logic: we make sure that `outputAmount` is at least our previously-defined cutoff amount. However, if the simulated output amount is higher, then we set `outputAmount` to expect that much, which protects us from slippage in case other transactions in the block happen to trade on the same pair. This function then sends our bundle to MEV-Share. If the bundle was received successfully, we should see a bundle hash logged to our console.

Using `getBuyTokenAmountWithExtra`, we define `outputAmount`, the amount we expect to receive from the trade. We create our backrun transaction `backrunSignedTx` and send it to MEV-Share in a bundle by calling `mevshare.sendBundle`.

*Real quick, let's break down the bundle we passed to `sendBundle`:*
```

The `block` parameter in `inclusion` specifies which block we want the bundle to land in. We indicate here that we want our bundle to land in the next block.

The `body` parameter is where we set our bundle's transactions. The order in which they're specified is the order in which they'll execute on chain. Each transaction is specified as an object, with either a `hash` parameter, or a `tx` parameter (paired with `canRevert` to specify whether this transaction is allowed to revert and land on chain). The transaction we specify with `hash` is the pending transaction from the event stream that we want to backrun. We have to use its hash because the MEV-Share event stream does not reveal the entire signed transaction. Naturally, the following transaction, specified by `tx`, is our trade.

Once we stitch all these new functions into our main loop, our bot will be done!

Sending a backrun bundle

When we detect a new pending transaction in the `mevshare.on("transaction")` callback that affects the price of our target pair, we need to send a bundle using the `backrunAttempt` function. This bundle checks our target price and sets up our trade to get us the best price possible.

src/index.ts

tsx mevshare.on("transaction", (pendingTx) => { // ... // TODO: backrun the user tx if (!transactionIsRelatedToPair(pendingTx, PAIR_ADDRESS)) { console.log('skipping tx: ' + pendingTx.hash) return } console.log(`It's a match: ${ pendingTx.hash }`) const currentBlockNumber = await provider.getBlockNumber() backrunAttempt(currentBlockNumber, nonce, pendingTx.hash) })

We'll also want to set up a callback that watches for new blocks and retries previous backrun attempts. Our bundles only target one block, but Protect transactions (which make up the transactions in the event stream) are valid for 25 blocks from when they're received. This means that if our backrun wasn't successful before, we can try again up to 24 more times.

Add this code to your `main` function:

tsx let recentPendingTxHashes: Array<{ txHash: string, blockNumber: number }> = [] provider.on('block', (blockNumber) => { for (const recentPendingTxHash of recentPendingTxHashes) { console.log(recentPendingTxHash) backrunAttempt(blockNumber, nonce, recentPendingTxHash.txHash) } // Cleanup old pendingTxHashes recentPendingTxHashes = recentPendingTxHashes.filter((recentPendingTxHash) => blockNumber > recentPendingTxHash.blockNumber + BLOCKS_TO_TRY) })

And in your `mevshare.on` callback, add this piece at the end:

tsx recentPendingTxHashes.push({ txHash: pendingTx.hash, blockNumber: currentBlockNumber })

When you're done, your main function should look like this:

src/index.ts

```tsx async function main() { console.log('mev-share auth address: ' + authSigner.address) console.log('executor address: ' + executorWallet.address) const PAIR_ADDRESS = (await uniswapFactoryContract.getPair(SELL_TOKEN_ADDRESS, BUY_TOKEN_ADDRESS)).toLowerCase() await approveTokenToRouter(SELL_TOKEN_ADDRESS, UNISWAP_V2_ADDRESS) const nonce = await executorWallet.getNonce('latest') let recentPendingTxHashes: Array<{ txHash: string, blockNumber: number }> = []

mevshare.on('transaction', async ( pendingTx: IPendingTransaction ) => {
  if (!transactionIsRelatedToPair(pendingTx, PAIR_ADDRESS)) {
    console.log('skipping tx: ' + pendingTx.hash)
    return
  }
  console.log(`It's a match: ${ pendingTx.hash }`)
  const currentBlockNumber = await provider.getBlockNumber()
  backrunAttempt(currentBlockNumber, nonce, pendingTx.hash)
  recentPendingTxHashes.push({ txHash: pendingTx.hash, blockNumber: currentBlockNumber })
})
provider.on('block', ( blockNumber ) => {
  for (const recentPendingTxHash of recentPendingTxHashes) {
    console.log(recentPendingTxHash)
    backrunAttempt(blockNumber, nonce, recentPendingTxHash.txHash)
  }
  // Cleanup old pendingTxHashes
  recentPendingTxHashes = recentPendingTxHashes.filter(( recentPendingTxHash ) =>
    blockNumber > recentPendingTxHash.blockNumber + BLOCKS_TO_TRY)
})

} ```

> For a full, working code example, check out https://github.com/flashbots/simple-limit-order-bot

*That's all you need!* This code will listen for new transactions and blocks, and trigger our code to send bundles when we find a transaction that changes the price.
```

Run the code and you should see something like this:

```
bash $ npx ts-node src/index.ts mev-share auth address: 0xE52A621a647A1013cE44EEBB37676F4c7205F87e executor address:
0x5b2c1E34C3Be923c123Ec858F474645B1Fcee0A3 skipping tx: 0x8ec57508495115338395a0de7cb9b85956845c83047bc523fc5b169fc9820251
skipping tx: 0xbe8be3ddfe27c03d79d44bf5f46a40e3491960c85fc29ceb9b9d7b968e3c7760 skipping tx:
0xd8bdceffde345bf71fb8058c0baae04eee58c25739dd31d3354c9777162be769 skipping tx:
0xfdc9ed3cd98e15d9e2b6bcea832f7f178d23f7c20d7900de5cf23af9051337c3 skipping tx:
0x4878de9bf339cffd8facb5037698839f97e8624806dec97f39d71254101ed712 skipping tx:
0x0b1964af41ddf4d4082f6d68073f200b1d72ffc9852441aa7621feba34c4e045 skipping tx:
0x23494341ec01dd3797778958e02cf9a8dbb8bb91989150b162cc7c16fe177267 skipping tx:
0xb201acbfe4c140a14cdd90b84d3a2c277d14bbb53e9bbacc095282c7b5e93c93 skipping tx:
0x8777978ea1c14af518d32ce74c8671905619ee8b6008c4c6f9a94e83fa2dda15 skipping tx:
0x6f25db574493009c6a9a22f3c15370038f80b8c86f3547f72811c0c58dfc6160 skipping tx:
0xb98f1e587f9a292043738f5c70e6532a6cd94a2f530a060556ce99256ca52e05 skipping tx:
0x0c347e08df1fee8bb0f4fafe606b1a1b8159804623bbcd78c03097dc316ca47a skipping tx:
0x2081950a40ca29cbc46c043f846a1e3205f3baaa09a8e531a20ff63c649617ed skipping tx:
0x8ccc554df02cf2c8880bde8ea4b7dcfecc533ec4c799e2a14a55282937934abf skipping tx:
0x7258bf29fb9879f10ee859ae80f491a7a652caf0825a1b2196cb2b39747e9a77 skipping tx:
0x8d49b3d723457845484c16b1d637265821a63c601dc0184417c2a20789ec2881 skipping tx:
0xfacd7b58b52184724fc1d8ad688544d0f45fb005110f49b790d320997e5ba79e skipping tx:
0x176d69b0353ea5cb421f7218f94ec54191bdca3cc1d659587546235d73ae8837 skipping tx:
0x518f3b9844a92d617146520b61face546aa399358f40136827fc24db8827b9ea skipping tx:
0x58da384fd342fdc8d8ba96faf3e12bc89417c7099888eaedfd77a00bcf10f701 skipping tx:
0x1431112c598a1a7969f048d6893b3ad72424fcc7a1de0b880ffc98585fcd2e7d skipping tx:
0x55bb0a2209665e464a385b188d0dcc1d91c93bae1509368cdc5abf9cb38afdfe skipping tx:
0x0978078667fd1bd834c5b8ca20e06b403e0df6311455de6dc6651c33542a594d skipping tx:
0xed23e27981fba7b4e661c10c9d4e70ada3eafb9cd4197a24f4566c1c5266e601 skipping tx:
0x3734c599be31a485a84376cf108fd76d5edc9637f834a9982a1600f0abd533d0 skipping tx:
0xc932320f77f8436037ab27ead89c004acc8f8bc52e0c19dcb6ecea103bc2973a skipping tx:
0xe880ce068eba29657bf14c535f421adefc43933fe84ef157cf4cdf7524edd22e skipping tx:
0x350aaae85d6167e219480fb60104d27a773675ff387c2634da44f1b36bbeba6b skipping tx:
0xfddcfb509e19be4500e4a1a03c97d77b2261fe41d8f019d48e0c0cea11cd1cc8 skipping tx:
0x08867976aecc2b09c57cf17d2e16dda9e3c3026da9c2ca3fed3187c7b577fa91
```

And after some time…

```
bash skipping tx: 0x82b46667867fd7e1684f14dd34503ee9b9bbe106352c00006592ceae8e58a9d7 skipping tx:
0x51c79454c4e86df176b51524fc254ab2e04c0861c329cdeab7a90d2e3309c3b7 skipping tx:
0xf0977236782c93462ea442514a28be64f11cd5feee647929b74834adff0b527a skipping tx:
0x8d5273f1eefa3a6daa40355df621e488dcf434e7bffb6201e0d98c26709ea35b skipping tx:
0x382ba4c5d62557277f494032b2578ba52435cccef7fd236f02278df337fa0f7b skipping tx:
0x04b40010c4a986f19fdedde1c2b0b92bf146074fae98693dc53a10d32a15bbba skipping tx:
0x7c89acc455e040f8690933af2fa40cac714f5f2790758972a35d4ea39ced32c9 skipping tx:
0xfcb49a980017cc3e0e5d5d4a0ff40aa170dc2e32021f81319c8d2b117b12bf52 skipping tx:
0x1861b5afc21a2e82dbdb8bc26777ced087c6a4830482b2f65273ba7159c204c8 skipping tx:
0x0817d834d6a537b6839858465f5e7ce2f95c9c5e353cf85282cd0f1da5b90e50 skipping tx:
0x387dd96be2b59da5f6e991b41d670f752fecd4d6571198729f1037996a4e2d3c skipping tx:
0x6b8577d861ce46f1e780c7ce59925ac9be40fd5428954c15a7e115cacaf974e2 skipping tx:
0xc41d53c886897aeda1fc401f337a2d8b3ef737e78f06a52cbbe29b501a1c7736 skipping tx:
0x52f2f383b4af41aa47f09ed8fd237358b32f33a0f471fc7be40b83179e513458 skipping tx:
0xbcc799ad42925730cbb724c0fe5ed7b8444e7fbc8a4219d7346a2a03e8ac5131 It's a match:
0xf76ce750b75ccd34b9174fdf8b574d129630d51422f3ca84f4f308952a4ea3b4
```

It may take a while to find a match — remember, the code scans for a trade on the ETH/DAI pair on Uniswap V2. There are lots of other events to consider for future improvements to this bot!

You may also consider removing the code that checks the event to see if it matches our pair address. It's possible that the price could move to our target level without us seeing an event to backrun. If we simply backrun every transaction we see, then we can potentially benefit from opportunities that we can't yet see. However, it is essential to understand how to use logs on MEV-Share, as they provide critical data that can be used for a multitude of other purposes, so we've introduced it here as a practical example.